CS 224 Advanced Algorithms — Spring 2017

Problem Set 3

Due: 11:59pm, Monday, February 27th

Submit solutions to Canvas, one PDF per problem:
`https://canvas.harvard.edu/courses/21996`

Solution max page limits: One page each for problems 1, 2, and 4, and two pages for problem 3

See homework policy at `http://people.seas.harvard.edu/~cs224/spring17/hmwk.html`

Problem 1: Consider splay trees. For any access sequence $\sigma = (x_1, x_2, \ldots, x_m)$ for each $i \in \{1, \ldots, n\}$ and fixed binary search tree T, let $C_T(\sigma)$ denote the cost of servicing σ with T. Let $S(\sigma)$ be the cost of servicing σ with a splay tree. We showed in class that $S(\sigma) = O(m + n^2 + C_T(\sigma))$.

(a) (7 points) Modify the weight function we used in class to show that in fact $S(\sigma) = O(m + n \log n + C_T(\sigma))$. As in the analysis in class, your proof should not use the fact that the optimal tree achieves the entropy bound.

(a) (3 points) Deduce that if each $i \in \{1, \ldots, n\}$ appears in σ at least once, then $S(\sigma) = O(m + C_T(\sigma))$.

Problem 2: (10 points) Define the Fibonacci numbers by $F_0 = 0, F_1 = 1$, and $F_k = F_{k-1} + F_{k-2}$ for $k > 1$.

(a) (3 points) Prove that for any integer $k \geq 0$, $1 + \sum_{i=0}^{k} F_i = F_{k+2}$.

(b) (7 points) Prove that for any node in a Fibonacci heap (not necessarily a root) with k children, the size of its subtree including the node itself is at least F_{k+2}. Thus, in particular, any top-level tree in the heap of rank k has size at least F_{k+2}. **Hint:** I recommend induction on something other than k.

Problem 3: (10 points) In Fibonacci heaps, when a node x loses 2 children, the subtree rooted at x is cut from x's parent and becomes a new tree in our top level forest. Suppose that instead we cut x's subtree away from its parent only after x loses k children.

(a) (5 points) Show that the amortized cost of decrease key is reduced as k increases. How does it decrease as a function of k? Note decrease key already has amortized cost $O(1)$ when $k = 2$, so the point here is just that the constant inside the big-Oh improves. **Hint:** modify the potential function from class.

(b) (5 points) Which operation(s) increase in amortized cost due to this change? Give a new bound as a function of k.

Problem 4: (10 points) You may remember the "disjoint forest" data structure for solving the union-find problem from your undergraduate algorithms course. If not, in the union-find problem we maintain a partition $\mathcal{C}$ of $\{1, \ldots, n\}$. We should support two operations:

- UNION(i, j): let $S \in \mathcal{C}$ be the partition containing i and $T \in \mathcal{C}$ the one containing j, and remove both S and T from $\mathcal{C}$ and add $S \cup T$ to $\mathcal{C}$ in their place.
- FIND(i): return any element in the partition $S \in \mathcal{C}$ that contains i, *however*, our data structure must obey the property that if i and j are in the same partition S, then FIND(i) and FIND(j) must return the same value.

One way to solve the above union-problem is to use the *disjoint forest* data structure. This data structure maintains a forest of rooted trees (not necessarily binary!). The nodes correspond to the elements $\{1, \ldots, n\}$. Each tree is a set in the partition. For any given tree, the root is the element which is returned during a FIND for any element in that tree.

Algorithm FIND(x):
1. **if** parent$[x]$ is `NULL`, then **return** x
2. **else return** FIND(parent$[x]$)

Algorithm UNION(x, y):
1. $x \leftarrow$ FIND(x)
2. $y \leftarrow$ FIND(y)
3. **if** $x \neq y$, then parent$[x] \leftarrow y$

We can see that the running time of FIND is the depth of x in its tree, which can be quite bad (it is not hard to do a sequence of UNIONs that cause some tree to be very imbalanced: even a path!). To remedy this issue, one simple heuristic is *path compression*. When we do a FIND on some node x, note we touch all of x's ancestors in its tree before reaching the root r: that is, we touch x, then x's parent p_1, then p_1's parent p_2, etc., until we touch some level-t ancestor $p_t = r$. With the path compression heuristic, after executing FIND(x), we then change the parent pointers of x as well as all the $p_1, \ldots, p_{t-1}$ to now point directly to r.

Algorithm FIND(x):
`// with path compression`
1. **if** parent$[x]$ is `NULL`, then **return** x
2. **else**
   (a) $r \leftarrow$FIND(parent$[x]$)
   (b) parent$[x] \leftarrow r$
   (c) **return** r

Prove that the amortized costs of UNION and FIND with path compression are both $O(\log n)$. **Hint:** use the same potential function as for splay trees with $w(x) = 1$ for each x (though the intended analysis is not at all related to that for splay trees, and is much more intuitive in this case!). **Note:** for those familiar with the "union-by-rank" heuristic, note that we are *not* using it here!